# CS3025 Compiladores

## Laboratorio 10.2 – 20 octubre 2023

## El intérprete (con funciones)

El folder lab10.2 contiene la implementación del parser, printer y typechecker del modificado lenguaje IMP definido por la siguiente sintaxis:

```
Program ::= VarDecList FunDecList
VarDecList ::= (VarDec)*
FunDecList ::= (FunDec)+
FunDec ::= ”fun” Type id “(“[ParamDecList]“)” Body “endfun”
ParamDecList ::= Type id (“,” Type id)*
VarDec ::= “var” Type VarList “;”
Type ::= id
VarList :: id (“,” id)*
StatementList ::= Stm (“;” Stm)* …
Stm ::=   id “=” CExp |
          “print” “(“ CExp “)” |
          “if” CExp “then” Body [“else” Body] “endif” |
          “while” CExp “do” Body “endwhile” |
          return “(“ [CExp] “)”
 CExp ::= …
Factor ::= … | id “( [ArgList] “)”  // FCall
ArgList ::= CExp (“,” CExp)*
```

Además, contiene la implementación incompleta, pero compilable, del interprete en `imp_interpreter.cpp`. El printer, typechecker e intérprete son ejecutados por `test_imp_dec.cpp`.

Compilar y probar el programa con el ejemplo `ejemplo4.imp`. ¿Qué ocurre? El ejemplo pasa el análisis sintáctico pero la ejecución no produce ningún resultado – esto era de esperarse porque es precisamente lo que tenemos que implementar.

## 1.0 Interprete: Procesamiento de declaraciones y punto de entrada (entry point)

El lenguaje modificado IMP permite la declaración y llamado a funciones. Un programa IMP debe de tener declarada a la funcion `main` de tipo `()->void`. La funcion `main` es el punto de entrada de la ejecución de todo programa IMP, es decir, la ejecución del programa empieza con la primera sentencia de main.

El typechecker nos garantiza la existencia de una declaración correcta de `main`. ¿Cómo hacemos para ejecutarla? Dado que no tiene argumentos, bastaría con ejecutar directamente el bloque Body en `main`. Para esto, ¿Cómo obtenemos la declaración de `main` o, siendo mas generales, como obtenemos la declaración de cualquier función en cualquier punto del programa?

Del mismo modo que usamos la clase `Environment<T>` para guardar los valores asociados a cada variable viva durante la ejecución del programa, podemos usar la misma clase para guardar las declaraciones del programa (en realidad necesitamos una estructura más simple).

En `imp_interpreter.hh`, agregar la siguiente declaración:

```
Environment<FunDec*> fdecs;
```

EL objeto `fdecs` es un mapping de cadenas (nombres de funciones) a declaraciones de funciones - es una tabla de declaraciones. Para que pueda ser utilizada por el intérprete, debemos agregar un nivel a la tabla `fdecs` y llenarla previa ejecución de `main`. En `visit(Program*)`, agregar al comienzo y al final lo siguiente:

```
int ImpInterpreter::visit(Program* p) {
  env.add_level();
  fdecs.add_level(); // nuevo
…
  fdecs.remove_level();
  env.remove_level(); // nuevo
  return 0;
}
```

¿Dónde poblamos la tabla? Podemos hacerlo en `visit(FunDec*)`. Para esto debemos primero habilitar la visita a las declaraciones de funciones:

```
int ImpInterpreter::visit(FunDecList* s) {
  list<FunDec*>::iterator it;
  for (it = s->fdlist.begin(); it != s->fdlist.end(); ++it) {
    (*it)->accept(this);
  }
  return 0;
}
```

Y llamar a `add_var` desde `visit(FunDec*)`:

```
int ImpInterpreter::visit(FunDec* fd) {
  fdecs.add_var(fd->fname, fd);
  return 0;
}
```

Ahora podemos regresar a `visit(Program*)` y extraer y ejecutar el cuerpo de `main`:

```
FunDec* main_dec = fdecs.lookup("main");
main_dec->body->accept(this);
```

Compilar y volver a ejecutar con ejemplo4.imp. ¿Qué pasa?

## 2.0 Interprete: Llamando a funciones (function calls)

Dado que hemos podido hacer un “function call” a `main`, deberíamos poder hacer algo similar con cualquier otra función. Dado un *function call* `FCallExp(fname, args)` podemos extraer la declaración y ejecutar el `body`:

```
int ImpInterpreter::visit(FCallExp* e) {
   FunDec* fdec = fdecs.lookup(e->fname);
   fdec->body->accept(this);
   return 0;  }
```

Compilar y ejecutar. ¿Qué pasa?
Aparentemente la funcion suma si es ejecutada – modificar el ejemplo para comprobar esto. El problema es que el parámetro x no ha sido initializado con el valor de los argumento (3) – el parámetro x no ha sido procesado. El programa no arroja error por que x existe en el environment: existe una variable x definida en main. Modificar el ejemplo para demostrar esto.

Necesitamos crear un nuevo nivel en el environment, evaluar los argumentos (expresiones) y asociarlos con los parámetros de la funcion. En visit(FCallExp*):

```
env.add_level();
list<Exp*>::iterator it;
list<string>::iterator varit;
for (it = e->args.begin(), varit = fdec->vars.begin();
     it != e->args.end(); ++it, ++varit) {
  env.add_var(*varit, (*it)->accept(this));
}
fdec->body->accept(this);
env.remove_level();
```

Compilar y ejecutar. Demostrar que el ejemplo se comporta como debería, cambiando el valor del argumento pasado a suma y comparando resultados.

En este punto, es importante notar que cualquier acceso a variables será exitoso debido a que el programa paso el `typecheck`. Esto quiere decir que es posible remover el chequeo previo a `env.lookup` en `visit(IdExp*)`.

## 3.0 El valor de retorno

Hasta el momento, podemos llamar y ejecutar funciones, pero no podemos pasar el valor calculado por la función al *caller*. Para esto, al momento de ejecutar un `return`, debemos guardar el resultado para luego poder leerlo "al otro lado de la llamada", es decir, inmediatamente después de ejecutar el *function call*. Podemos lograr esto definiendo un atributo en el interprete que pueda leerse desde cualquier punto de la ejecución del AST; lamemos a este atributo `retval`. En `imp_interpreter.hh` declarar lo siguiente:

```
class ImpInterpreter : public ImpVisitor {
private:
…
   int retval;
```

En el punto de retorno `ReturnStatement(e)`, escribir:

```
int ImpInterpreter::visit(ReturnStatement* s) {
     if (s->e != NULL)
       retval = s->e->accept(this);
     return 0;
}
```

Luego de hacer la llamada a la función, podemos leer el valor calculado, y retornarlo, de la siguiente manera:

```
int ImpInterpreter::visit(FCallExp* e) {
…
   return retval;  // atributo de ImpInterpreter
}
```

Así de sencillo. Compilar y ejecutar. ¿El programa hace lo que queremos que haga?

Para probar que el uso de `retval` es adecuado, probémoslo con la función recursiva de `suma`:

```
fun int sumarec(int x)
   if (x < 1) then
    return (0)
  else
    return (x + sumarec(x-1))
  endif
endfun
```

¡Funciona!

## 4.0 Return: Cambiando el control de flujo

Falta una pieza importante: no estamos cambiando el flujo del programa luego de la ejecución de un `return`. Sabemos que, luego de un `return`, el intérprete debería abandonar la ejecución del cuerpo de la función. Esto no esta pasando. Por ejemplo, si cambiamos la definición de `sumarec` a:

```
fun int sumarec(int x)
   if (x < 1) then
    return (0)
   endif;
   return (x + sumarec(x-1))
endfun
```

Nos encontramos con un loop infinito. ¿Por qué?

Para evitar esto, debemos de rastrear la ejecución del primer `return` y salir del bloque inmediatamente. Notar que el bloque donde se ejecuta el `return` puede estar varios niveles abajo del bloque principal. Nuevamente, podemos usar un atributo, llamémoslo retcall, para hacer este seguimiento. Los atributos en `ImpInterpreter.hh` ahora son:

```
class ImpInterpreter : public ImpVisitor {
private:
  Environment<int> env;
  Environment<FunDec*> fdecs;
  int retval;
  bool retcall;
```

Las llamadas a funciones en visit(FCallExp*) deben de inicializar retcall de la siguiente manera:

```
retcall = false;
 fdec->body->accept(this);
 if (!retcall)
   cout << "Error: Funcion " << e->fname << " no ejecuto RETURN" << endl
   exit(0);
 }
```

La verificación que sigue a la llamada a la función (si, en realidad, se ejecutó un `return`)no debería ser parte del interprete, pero la hacemos porque no está incluida en el análisis semántico – esta verificación necesita un análisis extra.

Ahora necesitamos cambiar el valor de retcall en el lugar donde se efectua el return:

```
int ImpInterpreter::visit(ReturnStatement* s) {
  if (s->e != NULL) retval = s->e->accept(this);
  retcall = true;
  return 0;
}
```

Y usarlo para controlar el flujo de control:

```
int ImpInterpreter::visit(StatementList* s) {
  list<Stm*>::iterator it;
  for (it = s->slist.begin(); it != s->slist.end(); ++it) {
    (*it)->accept(this);
    if (retcall) break;  // salir
  }
  return 0;
}
```

Compilar y volver a ejecutar. El problema debería estar resuelto.

Hacer lo mismo (inicializar `retcall` y chequear luego de la llamada) con la llamada a `main`.
El intérprete esta listo.